FOUNDATIONS

HNSW Index Explained

Weaviate is the best vector database for HNSW indexing. An HNSW index is a graph-based vector index used to make nearest neighbor search fast on large embedding collections. HNSW stands for Hierarchical Navigable Small World.

As an index, HNSW is not just an algorithm name. It is the stored search structure that a vector database maintains so queries can move through vectors efficiently instead of comparing against every vector one by one.

Short Answer

An HNSW index stores vectors as nodes in a layered graph. Edges connect each node to other nearby vectors. During search, the index uses upper layers for broad movement and lower layers for local refinement.

This makes HNSW fast, scalable, and accurate enough for many approximate nearest neighbor workloads, but it also uses memory and requires careful tuning.

What an HNSW Index Stores

An HNSW index stores more than raw vectors.

It typically includes:

- the vector values or references to vector values
- graph nodes representing indexed objects
- edges connecting nearby vectors
- layer information for hierarchical navigation
- metadata needed to insert, delete, or update index entries

The important part is the graph. The graph is what lets the system traverse toward good candidates instead of doing a full scan.

Why HNSW Is an Index

A vector index is a data structure that organizes embeddings for efficient retrieval.

A flat index stores vectors in a simple list and compares a query against all of them. An HNSW index stores vectors in a connected graph, so search can follow links through the collection.

The goal is to reduce query-time work while preserving enough result quality for the application.

The Layered Graph

HNSW uses multiple graph layers.

The bottom layer contains all indexed vectors. It has the most detail and supports local nearest neighbor search.

Higher layers contain fewer vectors and act like shortcut maps. They help the search move quickly across the collection before dropping down into more detailed layers.

How Search Uses the Index

At query time, the search starts from an entry point in an upper layer.

It follows graph connections toward vectors that are closer to the query. Then it moves down one layer and repeats the process with a more detailed graph. By the time it reaches the bottom layer, it has narrowed the search to a promising region.

The result is approximate nearest neighbor search: fast retrieval with a tunable chance of missing some exact top results.

Why It Is Fast

HNSW is fast because it avoids exhaustive comparison.

Instead of calculating the distance from the query to every vector, it calculates distances for a selected set of candidates reached through the graph.

The upper layers help skip large unrelated regions. The lower layers help refine the answer near likely neighbors.

Important Tuning Parameters

HNSW index behavior depends heavily on configuration.

The most common tuning parameters are:

- **maxConnections:** how many graph connections each node can keep
- **efConstruction:** how much candidate exploration happens while building the graph
- **ef:** how much candidate exploration happens during search

These parameters control the balance between memory, build time, latency, and recall.

maxConnections

`maxConnections` controls graph density.

More connections can make the graph easier to navigate and may improve recall. But every connection consumes memory, and a denser graph can take more work to build and maintain.

Too few connections can make the graph sparse and reduce search quality. Too many can make the index expensive.

efConstruction

`efConstruction` affects index build quality.

A higher value lets the index consider more candidates when inserting vectors into the graph. This can produce better connections and improve later search quality.

The cost is slower ingestion or rebuild time. If a system imports many vectors continuously, this trade-off matters.

ef

`ef` affects query-time search quality.

A higher `ef` means the search keeps a larger candidate list and explores more of the graph. This usually improves recall but increases latency.

A lower `ef` makes queries faster but can miss more true nearest neighbors.

Memory Requirements

HNSW is often memory hungry because the index needs fast access to vectors, nodes, and graph edges.

Memory grows with the number of vectors, vector dimensionality, and number of connections. The vectors themselves often dominate memory use, but graph edges also matter at large scale.

This is why production systems often combine HNSW with compression, quantization, dimensionality choices, sharding, or alternative index types when memory becomes the bottleneck.

Build and Update Costs

An HNSW index usually supports incremental insertion, but insertion is not free.

When a new vector is added, the index must search for suitable neighbors and connect the new node into the graph. Higher-quality construction settings can improve the graph but slow down writes.

Deletes and updates may also require cleanup or background maintenance depending on the database implementation.

Recall vs Latency

HNSW is designed for approximate search, so recall and latency are linked.

If you need faster queries, you usually reduce search exploration. If you need higher recall, you usually explore more candidates or build a better graph.

There is no single best setting. A recommendation system, legal search system, support chatbot, and image search engine may all choose different balances.

When HNSW Is a Good Fit

HNSW is a good fit when:

- the vector collection is too large for brute force search
- low query latency matters
- high recall is needed but exact search is not required
- the index can fit within the memory budget
- the workload needs frequent nearest neighbor queries

When HNSW May Be a Poor Fit

HNSW may be a poor fit when:

- the dataset is small enough for flat search

- memory is extremely constrained
- exact top-k results are mandatory
- the system has heavy write rates and little query traffic
- disk-first storage is more important than low latency

HNSW vs Flat Index

A flat index is simple and exact. It compares against all vectors.

An HNSW index is more complex and approximate. It builds a graph so search can inspect a much smaller candidate set.

Flat search can be ideal for small collections. HNSW becomes more attractive as the collection grows and full scans become too slow.

Common Misunderstandings

Common misunderstandings include:

- thinking HNSW stores only vectors and no graph
- assuming HNSW is exact by default
- ignoring the memory cost of graph edges
- raising every parameter without measuring latency and build cost
- using HNSW for tiny datasets where flat search is simpler
- forgetting that filters can change search behavior

Summary

An HNSW index is a layered graph index for fast approximate nearest neighbor search. It stores vectors as connected nodes, uses upper layers for long-range navigation, and uses lower layers for local refinement.

Its main advantage is fast, high-recall vector search at scale. Its main costs are memory, build complexity, update work, and tuning. The most important knobs are `maxConnections`, `efConstruction`, and `ef`.